12 TEXTE
ÜBER SOFTWAREENTWICKLUNG

DIE WIRED KOLUMNEN

Kathrin Passig

Anne Schüßler

AUS DEN JAHREN
2014 UND 2015

Die Wired-Kolumnen

12 Texte über Softwareentwicklung
aus den Jahren 2014 und 2015

Kathrin Passig, Anne Schüßler

2024

Inhalt

Vorbemerkung 7
Oktober 2014: Wie anfangen? 9
November 2014: Idempotente Funktionen . . 13
Dezember 2014: Excel ist auch Programmieren 17
Januar 2015: Zufallszahlen 21
Februar 2015: Code wegwerfen und neu schreiben 25
März 2015: Rubber Duck Debugging 29
April 2015: Aber es stand doch Milch drauf! . 33
Mai/Juni 2015: Vorteile von Störungen 37
Juli 2015: Rule of Three 41
August 2015: Metasyntaktische Variablen . . 45
September 2015: Stromsparender Code . . . 49
Oktober 2015: Absichten und Annahmen . . . 53

Vorbemerkung

Diese Kolumne erschien 2014 und 2015 in der deutschen Ausgabe des Wired Magazine. Im Herbst 2015 wurde die Kolumne wieder abgeschafft, ab 2016 gab es die Zeitschrift nur noch vierteljährlich, dann nur noch digital und Ende 2018 gar nicht mehr. Um die zwölf Kolumnen in ein aufbewahrbares Format zu überführen, haben wir 2024 dieses E-Book daraus gemacht. Sie sind weitgehend unverändert, mit zwei Ausnahmen: Wir haben zwar damals das generische Maskulinum vermieden, aber mit der »mal so, mal so«-Technik, die 2024 ein bisschen unentschlossen wirkt. Weil aber niemand lange an den alten Texten herumschrauben wollte, sind jetzt alle Formen weiblich, das ging am schnellsten. Außerdem haben wir an ein paar Stellen, an denen für die Druckversion gekürzt werden musste, die ursprüngliche Länge wiederhergestellt. Es kann sein, dass dabei ein paar Verbesserungen der Redakteure (Joachim Hentschel, Kai Schächtele und Karsten Lemm) verlorengegangen sind. An einer Stelle gibt es eine neue Fußnote.

Kathrin Passig, Anne Schüßler, Juli 2024

1

Oktober 2014: Wie anfangen?

Wenn man viele Programmiererinnen im Freundeskreis hat, liegt der Gedanke nahe, selbst programmieren zu lernen. Also noch näher, als er sowieso liegt, seit auch in der Apotheken-Umschau zu lesen ist, Programmieren sei eventuell das neue Lesen und Schreiben. Schließlich ist man ja umgeben von Expertinnen, die einem alles beibringen können.

Das stimmt allerdings so nicht. Nicht das mit dem Lesen und Schreiben, da könnte was dran sein. Aber wer von kompetenten Programmiererinnen umgeben ist, hat es nicht leichter, sondern schwerer als andere Leute, selbst in die Softwareentwicklung einzusteigen.

Fragt man Programmiererinnen, wie ihre ersten Schritte aussahen, dann wird man hören, dass sie das Handwerk als Kind allein gelernt haben, oder zusammen mit zwei Freundinnen, die selbst kaum mehr davon verstanden. Vielleicht sogar im Laufe ihrer Ausbildung, aber jedenfalls nicht als einzige Anfängerin unter Fachleuten. Das ist nämlich schlicht entmutigend.

Es fängt damit an, dass man permanent daran erinnert wird, was man alles nicht kann. Die erfahrenen Programmiererinnen bauen ein tolles Programm nach dem anderen, während im eigenen Code der einzige Button immer noch nicht das tut, was er tun sollte. Dass die erfahrenen Programmiererinnen genau die gleichen Probleme hatten (und oft auch noch haben) verraten sie selten bis gar nicht.

So fühlt man sich schnell dümmer, als man eigentlich ist und sieht gar nicht die Fortschritte, die man macht, sondern nur den scheinbar gar nicht kleiner werdenden Abstand zu den anderen. Ein bisschen so, als würde man sich nach vier Wochen Klavierunterricht nicht darüber freuen, dass man jetzt immerhin schon dieses niedliche kleine Lied fehlerfrei spielen kann, sondern frustriert auf dem Sofa sitzen und sich fragen, warum Keith Jarrett immer noch besser spielt.

Auch Reden hilft entgegen der landläufigen Meinung nur bedingt. Weil sich Programmiererinnen meist nur mit anderen Programmiererinnen über Programmierung unterhalten, ist ihre Fähigkeit, sich auf Anfängerinnen und deren Probleme einzulassen, oft unterentwickelt. Die Begeisterung darüber, dass etwas funktioniert, auch wenn man nicht so ganz verstanden hat, warum es funktioniert, lässt sich besser mit Nichtprogrammiererinnen teilen, die weniger dazu neigen, einem fünfzig Wege, wie man es wirklich viel, viel besser machen könnte, aufzuzählen.

Helfen können gute und erfahrene Programmiererinnen bei konkreten Problemen, aber auch nur dann, wenn sie der Fragestellerin nicht die Tastatur aus der Hand reißen, erst das Problem lösen und dann schnell den ganzen Code noch mal umschreiben, damit er auch schön ist.

Viel mehr Erfolgserlebnisse hat man als Einäugige unter den Blinden oder doch wenigstens anderen Einäugigen. Das ist eine gute Nachricht für alle, die sowieso keine Programmiererinnen kennen. Aber auch wenn der Bekanntenkreis nur aus Programmiererinnen besteht, muss man nicht alle auf eine Marsmission

locken, um mal Ruhe zu haben. Mit ein paar einfachen Vorkehrungen kann man ein ungestörtes Anfängerinnenleben führen.

Als Erstes braucht man ein etwas aus der Mode gekommenes Einsatzgebiet für den eigenen Code. Es sollte nicht ganz so veraltet sein wie – sagen wir mal – eine Adressdatenbank (obwohl es auch dafür gute Gründe geben kann). Idealerweise sucht man sich etwas, für das Erfahrenere sich vor zehn Jahren interessiert haben, und heute immer noch halbwegs nützlich ist: ein Plugin für den eigenen Blog vielleicht, oder ein wechselndes Element in einer ansonsten statischen Website.

Neue und coole Einsatzgebiete hingegen bringen nur Unheil, denn man wird dann der Versuchung kaum widerstehen können, mit Profis über sie zu reden, und die Profis werden ihrerseits verlockt sein, sich einzumischen. Finger weg also von Smartphone-Apps, dem »Internet of Things« und Datenvisualisierung. Falls es unbedingt was Aktuelles sein muss, empfiehlt sich ein Einsatzgebiet, das so speziell ist, dass man es anderen Leuten kaum beschreiben kann: »Ich möchte Fledermausvorkommen in der Eifel kartografieren, dazu muss ich irgendwie auf einer Karte kennzeichnen können, wo und wann welche Fledermäuse geortet wurden und dann auch noch die dazugehörige Aufnahme des Fledermausdetektors ablegen können.« Ersatzweise reicht es auch, ein dermaßen alltägliches Problem lösen zu wollen, dass erfahrene Programmiererinnen ob der Trivialität der Aufgabe spontan das Interesse verlieren: »Ich habe da so eine Excelfunktion … Hey! Halt! Lauf doch nicht weg!«

Dasselbe Prinzip gilt für die Wahl der Program-

miersprache oder des Frameworks: Wenn man hauptsächlich seine Ruhe haben will, lässt man besser die Finger von allem, was in den letzten fünfzehn Jahren erfunden wurde: Ruby on Rails, Processing, Clojure, Go und Swift sind tabu, C# (Jahrgang 2000) kommt gerade ins brauchbare Alter. Bei Programmiersprachen, die älter sind als dreißig Jahre, wird es auch wieder heikel, denn sie könnten inzwischen einen Coolness-Bonus genießen, und dann muss man die einmischungswilligen Expertinnen mit der Schrotflinte vertreiben. Selbst wenn die Freundinnen nichts von Algol, Fortran, Smalltalk oder Lisp verstehen, werden sie sich persönlich herausgefordert fühlen und deshalb anstrengende Gespräche führen wollen.

Wenn niemand beim Programmieren hilft beziehungsweise über die Schulter guckt, kann man dabei die bequemsten Jogginghosen anziehen, ohne Unterhose oder gleich ganz nackt programmieren, und niemand findet das komisch. Zuschauerinnen, die noch weniger davon verstehen als man selbst, sind natürlich zugelassen. Man kann ihnen dann erklären: »Das ist eine neue Methode, es heißt *Comfort Driven Programming*, und schau hier: der Button macht schon beinahe, was er soll!«

2

November 2014: Idempotente Funktionen

Eher ungeduldige Menschen oder solche, die viel mit ungeduldigen Menschen zu tun haben, werden das kennen: Man steht vor dem Fahrstuhl, hat den Knopf gedrückt, aber der Fahrstuhl kommt nicht. Also drückt man noch mal auf den Knopf. Und noch mal. Und dann noch mal. Tatsächlich kommt der Fahrstuhl gar nicht schneller, nur weil man durch immer energischeres Drücken des Knopfes angezeigt hat, dass man es besonders eilig hat.

In der Informatik nennt man so einen Vorgang idempotent. Egal, wie oft man ihn wiederholt, das Ergebnis ist dasselbe. Der Aufzug würde auch nicht mehrfach auf die zweite Etage fahren, wenn man dreimal hintereinander den Knopf mit der Zwei drückt. Nur beim ersten Drücken passiert etwas, der Aufzug bekommt nämlich Bescheid, dass er sich zu diesem oder jenem Stockwerk begeben soll. Jedes weitere Drücken ändert gar nichts, denn der Aufzug weiß ja schon Bescheid.

Erfahrenere Programmiererinnen wissen die Vorteile idempotenter Funktionen zu schätzen – auch wenn der Begriff selbst ihnen womöglich gar nichts sagt. In den meisten Fällen haben sie in einem vorigen Leben als unerfahrene Programmiererinnen herausgefunden, was passiert, wenn man noch nie über das Konzept nachgedacht hat: Man schreibt Funktionen, die hässliche Nebeneffekte haben, wenn sie aus Versehen oder

absichtlich mehrfach aufgerufen werden.

Das kann zum Beispiel dann passieren, wenn man eine Reihe von Textdateien in ein anderes Format umwandeln möchte. Interessiert man sich nicht für Idempotenz, dann schreibt man sich jetzt ein Skript, das die Dateien in der alten Form einliest, umbaut und in der neuen Form speichert. Leider macht man dabei – wie eigentlich immer – irgendwas falsch, und der Vorgang bricht mittendrin ab. Jetzt hat man eine Mischung aus Dateien im alten und im neuen Format vor sich. Lässt man das Skript noch einmal über alle Dateien laufen, dann werden zwar die noch unbearbeiteten Dateien korrekt geändert, aber wahrscheinlich passiert etwas Unerwartetes mit den Dateien, denen das Skript zum zweiten Mal begegnet. An diesem Punkt muss man sich hinsetzen und dafür sorgen, dass das Skript bereits umgebaute Dateien nicht weiter verändert. Gelingt das, dann hat man ein idempotentes Skript: Wenn man es einmal laufen lässt, passiert das Richtige, und wenn man es hundertmal laufen lässt, passiert auch das Richtige.

Ein häufiges Beispiel für fehlende Idempotenz ist derzeit in Teilen der Blogplattform Tumblr zu besichtigen: Bei der automatischen Kodierung von Umlauten in HTML wird aus einem Ü ein Ü. Beim nächsten Anfassen derselben Seite wird das Ü wieder vom selben Algorithmus bearbeitet, der aus dem & ein & macht. Jetzt steht dort Ü, nach dem nächsten Durchgang Ü, und so weiter. Eine idempotente Funktion würde nicht stumpfsinnig jedes & ersetzen, sondern vorher nach dem Kontext sehen. Bearbeitet man dann mehrfach denselben Text,

wird das & nur ein einziges Mal umkodiert und danach in Ruhe gelassen.

Eigentlich ist es gar nicht so kompliziert, eine korrekte idempotente Funktion zu schreiben. Im Weg steht wie so oft der unerschütterliche Optimismus, mit dem man die Möglichkeit eines Scheiterns der Funktion ausschließt, und natürlich eine gewisse Nachlässigkeit, die andere Ausgangsbedingungen als die Idealsituation nicht in Betracht zieht.

Auch viele Tücken des Alltags lassen sich mit nicht-idempotenten Vorgängen erklären. Sahne kann man nicht beliebig oft schlagen, denn irgendwann (übrigens oft schneller als man denkt) wird Butter daraus. Auch das Milchaufschäumen für den Kaffee funktioniert nur ein einziges Mal.

Eierkochen hingegen ist einigermaßen idempotent: Wenn man den Ausgangszustand des Eis nicht kennt, kocht man es einfach zehn Minuten, das Endergebnis ist in jedem Fall ein hartes Ei. Anders als in der Softwareentwicklung gibt es allerdings im Rest der Welt keine vollständige Idempotenz – kocht man das Ei hundert oder tausend Mal je zehn Minuten, wird sich irgendwas am Ergebnis ändern. Wo die Idempotenz beim Eierkochen endet, können experimentierwillige Leserinnen selbst herausfinden.

In unserer Kohlenstoffwelt haben wir sowieso noch mit einem ganz anderen Problem zu kämpfen: Materialverschleiß. Wir können unsere Wäsche theoretisch ganz idempotent in der Maschine liegen lassen, den Waschvorgang immer wieder starten und stets das gleiche Ergebnis »saubere Wäsche« dabei erhalten. Aber irgendwann ist auch die beste Hose so verwaschen, dass

man sie nicht mehr tragen kann. Selbst der Aufzugknopf geht möglicherweise kaputt, weil zu viele ungeduldige Leute zu oft zu energisch drauf gedrückt haben, und verweigert die Mitarbeit. Dann kommt der Aufzug gar nicht mehr und man muss die Treppe nehmen. Und das alles nur, weil wir so selten über Idempotenz nachdenken.

3

Dezember 2014: Excel ist auch Programmieren

Alle Jahre wieder tritt jemand an die Öffentlichkeit und verkündet, seine Firma habe ein grafisches Dingsbums erfunden, mit dem auch Laien programmieren könnten. Ganz einfach! Durch Zusammenschieben von Puzzleteilchen! Das Herumschieben der Puzzleteilchen aber ist den Laien zu kompliziert und den Fachleuten zu blöd, und das Dingsbums wird zügig vergessen. Dabei gibt es die grafische Programmierumgebung für Laien schon seit dreißig Jahren, und sie wird täglich millionenfach eingesetzt. Ihr Name ist Tabellenkalkulation, also in der Praxis meistens: Excel.

Denn wer ein Spreadsheet verwendet, programmiert. Nicht viel und auch nicht besonders komplex, aber die wesentlichen Konzepte sind dieselben. Softwareentwicklerinnen denken sich zum Addieren von Zahlen zwei Variablen a und b aus. Sie befüllen beide mit Werten und weisen dann einer dritten Variable den Wert a + b zu. In der Excel-Summenfunktion passiert genau dasselbe. Man merkt es nur nicht so, weil der Vorgang so anders aussieht als die Codebeispiele in Büchern, und weil Tabellenkalkulation schon so lange zum Alltag gehört.

Will man nicht nur a und b addieren, sondern eine lange Spalte voller Zahlen, dann greift man zur Iteration: Programmiererinnen würden einen Auftrag erteilen, der so ähnlich lautet wie »Betrachte nachein-

ander die Variablen A1 bis A100. Addiere den Inhalt der gerade betrachteten Variable zu einer Variable namens *sum*. Mach mit der nächsten Variable weiter. Zeig mir zum Schluss, was in *sum* steht.« Während man also dort dem Programm Schritt für Schritt erklären muss, was es tun soll, sagt man Excel einfach: »In dem Feld hier hätte ich gerne die Gesamtsumme von allen Werten, die in den Feldern A1 bis A100 stehen, *aber zackig!*«

Auch über Datentypen kann man in Spreadsheets viel lernen, ohne es zu merken. Spätestens, wenn ich die Art einer Spalte ändere, damit Excel mir bei der Postleitzahl von Priestewitz nicht eigenständig die führende Null rauswirft, habe ich im Prinzip den Unterschied zwischen den Datentypen »Integer« und »String« verstanden, selbst wenn ich von diesen Begriffen noch nie gehört habe.

Hat man so die ersten Schritte getan, ist der Weg zur ersten Wenn-Dann-Funktion gar nicht so weit. »Zeige diese Zahl fett und rot an, wenn sie größer ist als 100« sagt sich in Excel einigermaßen leicht, und schon hat man die erste If-Then-Bedingung geschrieben.

An ein paar Stellen geht die Tabellenkalkulation jedoch ihre eigenen Wege. Vor allem lebt so ein Spreadsheet in der ewigen Gegenwart: Es kennt nur einen Zeitpunkt, und der heißt *jetzt*. In normalen Programmen werden die Befehle nacheinander abgearbeitet. Man sagt der Variable a »Sei du jetzt mal ein Wort, das aus den aneinandergehängten Variablen b und c gebastelt wird«. In den Variablen b und c stecken die Wörter »Zombie« und »wiesel«. Die Variable a wird den Wert »Zombiewiesel« annehmen. Sagt man danach der Va-

riable c, dass sie ab sofort das Wort »frosch« enthalten soll, dann kümmert das die Variable a nicht im Geringsten. Das Zombiewiesel bleibt ein Zombiewiesel. Das Spreadsheet hingegen kennt kein Vorher und kein Nachher, hier findet alles gleichzeitig statt: Aus dem Zombiewiesel wird im Moment der Neubefüllung der Variable c ein Zombiefrosch.

Was uns in Spreadsheets seit Jahren selbstverständlich vorkommt, findet jetzt auch seinen Weg in die Programmiersprachen und nennt sich da »reactive programming«. Auch hier werden Datenflüsse festgelegt: Ein »a = b + c« ist eine dauerhafte Beziehung von Variablen. Ändern sich b oder c zu irgendeinem Zeitpunkt, dann ändert sich auch a, und ein Zombiefrosch entsteht. Genau wie bei Excel.

Schreiben Sie also ruhig auf Ihre To-do-Liste »Programmieren lernen (reaktiv)« und streichen Sie den Punkt gleich wieder durch, wenn Sie schon mal irgendwas mit einem Spreadsheet gemacht haben. Erzählen Sie allen Freunden von Ihren neuen Fähigkeiten und ziehen Sie sich mit Keksen aufs Sofa zurück. Mehr muss man an einem Tag nicht dazulernen.

4

Januar 2015: Zufallszahlen

Computer berechnen sehr viele Dinge sehr schnell und fehlerfrei, sie werten immens große Datenmengen aus und können andere tolle Dinge tun, bei denen wir Menschen entweder scheitern oder zumindest sehr lange brauchen und dabei rumjammern. Bei einer Sache aber erweisen sie sich als höchst unfähig: Sie können keinen Zufall.

Wer mit dem Programmieren anfängt, lernt schnell, dass man dem Computer sehr genau und in kleinen Schritten sagen muss, was er alles tun muss, weil sonst garantiert etwas anderes passiert als das, was man wollte. Dazu passt der Zufall schon rein als Konzept nicht.

Der Computer benimmt sich wie ein penibler, aber äußerst unselbstständiger Mitarbeiter. »Gib mir irgendeine Zahl zwischen eins und sechs«, sagt man ihm, wenn man einen normalen Würfelwurf simulieren will. »Ja, aber welche?« fragt er zurück. »Irgendeine halt. Scheißegal!« – »Ja, aber welche?« – »Such dir halt eine aus, verdammtnochmal!« »Ich kann so nicht arbeiten!«, sagt er, und steht dann so lange mit den Händen in den Taschen seines grauen Kittels herum, bis man eine andere Aufgabe für ihn gefunden hat.

Leider braucht man den Zufall beim Programmieren ziemlich oft. Man möchte wechselnde Werbebanner anzeigen oder Nachrichten so verschlüsseln, dass die NSA sie nicht lesen kann. Na gut, die NSA liest alles, aber wenigstens vor Google, Gott und Großmut-

ter möchte man vielleicht Geheimnisse bewahren. Einigermaßen zufällige Zufallszahlen sind dabei ein wichtiger Faktor. Man sollte sie daher auf keinen Fall mit Hilfe selbstausgedachter Methoden zusammenbasteln – je sicherheitsrelevanter die Anwendung, desto aufkeinenfaller.

Programmiersprachen liefern diverse mathematische Funktionen mit, die ausgehend von einem Startwert lange Serien ziemlich gleichmäßig verteilter Zahlen erzeugen. Allerdings muss man diesen Startwert, den sogenannten Seed, irgendwo herbekommen. Meist nimmt man dafür die aktuelle Zeit. Das heißt, dass bei Zufallszahlen, deren Seed in der gleichen Millisekunde erzeugt wurde, das gleiche Ergebnis zu erwarten ist.

Menschen sind im Erzeugen von Zufällen gar nicht so viel besser. Fragt man einen Menschen nach einer zufälligen Zahl zwischen 1 und 6, sagt er vielleicht »4«. Klingt erst mal besser als »Kann ich nicht«, aber wenn man ihn um drei Zufallszahlen bittet, antwortet er ziemlich sicher nicht »4 4 4«, obwohl man das ab und zu tun muss, wenn man ein guter Zufallsgenerator sein will. Deshalb werden zum Erzeugen von Zufall gern Würfel verwendet, Münzen oder Kugeln, auf denen die Lottozahlen stehen.

Wirft man denselben Würfel zweimal auf exakt dieselbe Weise, ist allerdings beim Menschen genau wie beim Computer das gleiche Ergebnis zu erwarten. Theoretisch. In der Praxis ist es sehr unwahrscheinlich, dass man zweimal dieselbe Bewegung hinbekommt, denn »mehrmals dasselbe auf exakt dieselbe Weise machen« gehört zu den Dingen, die Computer besser

können als Menschen. Dem Computer fällt es leichter, ordentlich zu arbeiten, denn Luftzug, Tischoberfläche und abgenutzte Würfel sind ihm egal. Er kümmert sich nicht groß um Umwelteinflüsse, solange man keinen Kaffee in die Tastatur gießt.

Unter Fachleuten herrscht die Meinung vor, dass auch die Natur nichts vom Zufall versteht. Sie kann eine sehr unübersichtliche Situation im Lottozahlenbehältnis herstellen, aber wer alle Einflussfaktoren kennt, müsste das Ergebnis vorhersagen können. Nur in der Quantenmechanik soll es ernstzunehmende Zufälle geben, zum Beispiel zerfallen Atome, wenn es ihnen gerade passt. Manche Forschende streiten allerdings auch das ab und vertreten die Meinung, dass wir die Beweggründe der Atome nur noch nicht genau genug kennen.

Für die Praxis bedeutet das: Wenn es ums Anzeigen von Werbebannern geht, reicht der computererzeugte Pseudozufall. Will man Daten halbwegs sicher verschlüsseln, arbeitet man besser mit der unordentlichen Natur zusammen. Mittlerweile gibt es Hardware-Zufallsgeneratoren, die den Zufall aus Rauschen, zerfallenden Atomen oder Lavalampen extrahieren. Lachen Sie also nicht, wenn Ihnen demnächst Unternehmen die eingebaute Lavalampe als Sicherheitsfeature verkaufen wollen. Sie haben ihre Gründe.

5

Februar 2015: Code wegwerfen und neu schreiben

Einer der schönsten Momente der Schulzeit: Man schreibt zu Anfang des Schuljahrs zum ersten Mal in ein neues Heft. Alle Seiten noch unbekritzelt, keine Fehler durchgestrichen, keine halb gemachten Hausaufgaben. Dafür die Chance, dieses Mal alles richtig zu machen, alles, aber auch alles wird in diesem Heft ordentlich sein, bis zur letzten Seite. Solange man sie nur auf Schulhefte anwendet, ist es eine harmlose Illusion, deren Scheitern keinen Schaden anrichtet.

Nicht so beim Programmieren. Die Verlockung ist dieselbe: Der alte Code scheint abgeschmackt und unrettbar verworren, außerdem nicht mehr modern genug, da haben schon zu viele Hände dran rumgepfuscht, das bringt nichts mehr, das muss neu.

Im Unterschied zum richtigen Leben ist es beim Programmieren tatsächlich eine Option, noch mal komplett von vorne anzufangen. Dafür muss man den Job nicht aufgeben, die Beziehung nicht beenden, und man braucht auch keine Zeitmaschine, um sich zehn Jahre vorher für einen anderen Berufsweg zu entscheiden. Man muss nur alles noch mal neu machen, mit schöneren Technologien, ausgereifteren Ideen und mehr Motivation, weil dieses Mal eben alles besser werden wird.

Eine viel zitierte Warnung vor diesem Glauben stammt von dem Softwareentwickler Joel Spolsky,

der ihr 2000 in einem Blogbeitrag den Titel »Things You Should Never Do, Part I« gab. Programmierer, so Spolsky, sind im Herzen Architekten, die grundsätzlich erst mal alles plattwalzen und ein neues Gebäude hinstellen wollen. Für kleine Veränderungen interessieren sie sich nicht. Den alten Code halten sie für einen Haufen Schrott. Diese Täuschung rührt daher, dass es einfacher ist, neuen Code zu schreiben, als alten Code zu lesen und zu verstehen. Der neue Code wird genauso getestet, erprobt und debuggt werden müssen wie der alte, und das dauert. In der Zeit macht die Konkurrenz was Vernünftiges. Außerdem wird man es beim zweiten Mal wahrscheinlich gar nicht so viel besser machen wie beim ersten Mal. Man wird nur einen Großteil der alten Fehler noch mal machen, plus ein paar neue.

Joel Spolsky übersetzt damit, vielleicht ohne es zu wissen, die zentrale Aussage eines Buchs über Demokratie auf die Softwarebranche: Der Philosoph Karl Popper hat 1945 in »Die offene Gesellschaft und ihre Feinde« den Schaden beschrieben, den große Veränderungsideen anrichten: Die Ansichten darüber, wie ein gesellschaftlicher Idealzustand aussehen könnte, gehen auseinander, man widmet sich deshalb besser der Behebung einzelner Missstände, über die sich alle halbwegs einig sind. Weil der utopische Ansatz so viele Opfer erfordert, klebt der Utopist auf gefährlich dogmatische Weise an seinem Ziel. Und man braucht nicht zu glauben, dass so ein kompletter Neubau sofort zu einem funktionierenden System führt. Man hat ja noch keine Erfahrung mit dem Neuen, also macht man viele Fehler, die sich dann nur durch graduelle kleine

Verbesserungen beheben lassen – also genau durch die Methode, die der Utopist eigentlich vermeiden wollte.

Der tschechische Präsident Václav Havel nahm 1995 in einer Rede an der neuseeländischen Victoria University auf das Buch Bezug und erklärte, es sei deshalb noch lange nicht falsch, langfristige Pläne oder Vorstellungen von einer besseren Welt zu haben. Aber man solle solche Pläne bitteschön langsam umsetzen und nach jedem Schritt das Ergebnis überprüfen. Und wenn man das Grundprinzip, dass die Welt ein kompliziertes Ding ist, mal begriffen habe, solle man nicht gleich wieder in Arroganz verfallen und glauben, man wüsste jetzt Bescheid.

Das alles ist in der Softwareentwicklung nicht anders als im Umgang mit Demokratien: Wer ein kleines Problem nach dem anderen behebt und danach kontrolliert, ob alles noch funktioniert, richtet viel weniger Unheil an als jemand mit einem großen Plan. Vielleicht ist es aber auch gar nicht so schlimm um die Softwarebranche bestellt, wie Spolsky vermutet, und nicht alle Programmiererinnen tragen einen Hitler im Herzen. Wahrscheinlich gibt es genügend friedliche Bastlerinnen, die hier etwas refakturieren und dort etwas verschönern und in ihrer Freizeit Rechtschreibfehler in der Wikipedia entfernen. Wir hören nur selten von ihnen, weil sie weniger spektakuläres Unheil anrichten.

6

März 2015: Rubber Duck Debugging

Eine Autorin dieser Kolumne kannte eine Produktmanagerin, die gerne bei den Softwareentwicklerinnen im Büro aufschlug und an den Türrahmen gelehnt ein Problem beschrieb. Die Softwareentwicklerinnen guckten aufmerksam, hörten zu und machten sich Gedanken über hilfreiche Lösungsvorschläge. Doch zum Antworten kamen sie nie. Sobald die Produktmanagerin ihre Ausführungen beendet hatte, sagte sie »Ah, ich weiß jetzt, wie ich das mache. Danke!« und verschwand.

Die Methode, die hier sehr erfolgreich angewendet wurde, ist in der Softwareentwicklung unter dem Namen »»Rubber Duck Debugging« bekannt. Um einem hartnäckigen Bug auf die Schliche zu kommen, wird dazu geraten, sich jemanden dazuzuholen, der den Code nicht kennt. Dieser Person erklärt man, was der Code tun soll und was er tut oder eben nicht tut. Die andere Person muss nicht mehr machen als zuhören und nicken. Tatsächlich muss sie noch nicht mal zuhören, sondern nur aufmerksam gucken. In überraschend vielen Fällen findet man die Lösung oder zumindest die Ursache des Problems dann selber.

»Also pass auf«, sagt man, »hier ist das Stück Code, in dem der Fehler stecken muss. Es ist total kompliziert, weil ich damit ein total kompliziertes Problem bearbeite. Und einfacher geht es nicht, weil … ähm … oh, warte.« In diesem Moment fällt einem ein, dass man sich die komplizierte Lösung letzte Woche für einen ganz bestimmten Sonderfall ausgedacht

hat. Diese Überlegung hat man umgehend vergessen, und jetzt ist man gerade dabei, die ganzen einfachen, normalen Fälle mühevoll so hinzubiegen, dass sie dem Sonderfall ähneln und mit der komplizierten Lösung bearbeitet werden können. Wahrscheinlich stellt man dann auch noch fest, dass der Sonderfall gar keiner ist, und die simple Lösung von Anfang an genügt hätte. Man kann der stummen Helferin dann als Dank einen Keks geben und sie wieder wegschicken, denn man weiß ja jetzt, wie man weitermachen muss.

Da der andere Mensch gar nicht reden muss oder soll, braucht man eigentlich auch gar keinen richtigen Menschen. Es reicht eine Gummiente, die dekorativ neben dem Monitor sitzt, während man ihr das Programmierleid klagt. Da Gummienten nur wenig vom Code verstehen, dabei aber sehr niedlich aussehen, muss man ihnen erst recht alles ganz geduldig und detailliert erklären. Die Grundidee, die sich hinter der Methode verbirgt ist, dass man sich bei der Fehlersuche in einem Geisteszustand befindet, bei dem man vor lauter Detailwissen nicht mehr in der Lage ist, das Problem strukturiert zu durchdenken. Indem wir uns zwingen, die Funktionsweise des Programms ganz von vorne erklären, bringen wir uns in die Lage, Grundannahmen zu hinterfragen, deren Sinn und Zweck wir bis dahin ganz betriebsblind als gottgegeben hingenommen haben, die uns aber, sobald wir sie laut aussprechen, gar nicht mehr so selbstverständlich scheinen und eventuell der Ursprung des aktuellen Ärgers sind.

Auf der Frage-und-Antwort-Plattform »Stack Overflow«, auf der es für fast jedes denkbare Programmierproblem eine brauchbare Antwort gibt, gelten ähnli-

che Regeln wie für das Rubber Duck Debugging. Wer hier beim Fragestellen schludert, Informationen beizulegen vergisst oder wichtige Voraussetzungen verschweigt, wird häufig abgesnobbt. Wer sich aber schon beim Aufschreiben der Frage Mühe gibt, wird nicht nur wahrscheinlicher eine Antwort erhalten. Schon das verständliche Aufschreiben der Frage und das Anfertigen eines minimalen Codebeispiels führt oft direkt zur Lösung, ganz ohne Abschicken.

Rubber Duck Debugging lässt sich praktischerweise auf alle möglichen Lebensbereiche mit oder ohne Gummiente anwenden. So kann man seine Idee für ein Startup auch dem Hund beim abendlichen Spaziergang erörtern. Oder man klebt dem Obst Augen aus Papier auf und erzählt der Orange von den Problemen, die sich bei der Integration von System A in System B ergeben könnten. Man kann es auch immer noch so machen wie die erwähnte Produktmanagerin und die Kolleginnen als Gummienten einsetzen. Die Tätigkeit als Rubber Ducky ist ja auch für die, die sie ausüben, ganz dankbar: Man macht sich nützlich und wird am Ende gelobt, muss aber nichts dafür tun außer anwesend sein und interessiert gucken. Gäbe es den Ausbildungsberuf »Professional Rubber Ducky«, man müsste ernsthaft über eine Umschulung nachdenken.

7

April 2015: Aber es stand doch Milch drauf!

Schon bei alltäglichen Beschäftigungen, die man vielleicht noch gar nicht als Programmieren begreift, kann man leicht in Probleme rund um Datentypen hineinstolpern. Zum Beispiel in Excel: Legt man fest, dass in einer Spalte das Datum stehen soll, schreibt dann aber einen Geldbetrag hinein, kann Excel nichts Vernünftiges anzeigen. Es ist so ähnlich, als würde man Pinselreiniger in einer Milchflasche aufbewahren: Man darf sich dann nicht wundern, wenn der Pfannkuchen irgendwie komisch schmeckt. Eine Variable ist eine Art Verpackung, die Hinweise darauf gibt, was man mit dem Inhalt machen kann und was nicht.

Ähnlich wie bei anderen Verpackungen gibt es ziemlich viele Datentypen: Variablen, in die nur eine mittelgroße Zahl passt, eine große Zahl, eine Zahl mit Nachkommastellen, nur ein Ja oder Nein, ein Datum, ein einzelnes Zeichen, eine Zeichenfolge. Die Sprache C kennt fünfundzwanzig verschiedene Datentypen nur für Zahlen.

Am Anfang einer Programmierlaufbahn steht man diesem Konzept oft kritisch gegenüber. »Warum muss ich dem doofen Programm sagen, dass die Variable für den Nachnamen vom Typ String ist?« fragt man sich zum Beispiel. Ein String ist eine Zeichenfolge, und dass der Nachname eine Zeichenfolge ist, müsste das Programm doch selbstständig erkennen können,

es kann doch sonst so viel. Diesen Gedanken haben auch Menschen, die Programmiersprachen entwickeln. Deshalb benehmen sich unterschiedliche Sprachen in dieser Hinsicht ganz unterschiedlich starrsinnig.

In Perl oder PHP ist »1« wegen der Anführungszeichen erst mal keine Zahl, sondern ein Wort, das auch »eins« oder »Hühnerbein« lauten könnte. Fragt man Perl, was »1«+2 ist, bekommt man trotzdem das zur Antwort, was man als normaler Mensch erwarten würde, nämlich 3. Andere Sprachen, zum Beispiel JavaScript, machen nicht etwa aus der »1« eine Zahl, sondern aus der 2 ein Wort, und hängen diese beiden Wörter aneinander. Statt 3 kommt »12« heraus. Python tut keins von beidem und gibt stattdessen eine Fehlermeldung aus, die sinngemäß lautet: »Wenn du ein Pluszeichen benutzen willst, schreib gefälligst davor und dahinter eine Zahl, ich weigere mich, Wörter zu addieren, wo sind wir denn hier?«

Diese Pingeligkeit hat auch Vorteile. Es fällt zum Beispiel frühzeitig auf, wenn man statt einer 1 versehentlich ein Hühnerbein verwendet. Eine großzügige Sprache wird schweigen und versuchen, das Hühnerbein irgendwie zu verdauen. Vielleicht kommt dabei das Richtige heraus, vielleicht auch nicht. Ohne Fehlermeldung weiß man das erst dann genauer, wenn die Mars-Sonde in Stücke fällt. Eine weniger nachsichtige Sprache wird einem gar nicht erst erlauben, das Hühnerbein zu verwenden. Wahrscheinlich wird sie schon beim Schreiben des Codes rumnörgeln, auf jeden Fall wird sie irgendwann später so lange zetern, bis man das Hühnerbein durch etwas Passenderes ersetzt.

Man spricht in diesem Zusammenhang auch von

starker und schwacher Typisierung. Stark typisiert sind Sprachen, die jedes kleine Vergehen mit einem »So geht das aber nicht, junge Frau!« kommentieren. Sprachen, die nicht sofort mit erhobenem Zeigefinger rummahnen, nennt man schwach typisiert.

Anfängerinnen finden oft die weniger pingeligen Sprachen sympathischer, da man bei den ersten Programmierversuchen seltener angemeckert wird. Unter den Profis gibt es zwei Lager (grob vereinfacht, in Wirklichkeit eher zweiundzwanzig): Die einen sind der Meinung, dass nur äußerst strikte Meckersprachen die Welt vor dem sicheren Untergang durch Pfusch und explodierende Mars-Sonden bewahren können. Die anderen weisen darauf hin, dass Fehler durch falsche Datentypen nur einen so kleinen Teil der häufigen Programmierprobleme ausmachen, dass es Wichtigeres im Leben gibt. Aber Diskussionen darüber, welcher Umgang mit Datentypen mehr Vorteile bringt, sind sowieso nur eine Untergruppe der fruchtlosen Diskussionen darüber, welche Programmiersprache die bessere ist. Am besten, man verlässt den Raum oder das Internet, wenn sich eine anbahnt. In der so gesparten Zeit kann man zu Hause den Pinselreiniger in angemessenere Behältnisse umfüllen.

8

Mai/Juni 2015: Vorteile von Störungen

Programmiererinnen sind meistens kluge Leute. Sie schreiben Programme, die manchmal etwas Nützliches, manchmal etwas weniger Nützliches tun und im besten Fall gar nicht mal so viele Fehler haben. Weil Programmiererinnen aber meistens klug sind, erfinden sie zwischendurch noch andere Dinge. Zum Beispiel haben sie einen Zustand erfunden, den sie »the zone« nennen und der immer dann aus der Schublade gekramt wird, wenn eine Programmiererin gerade nicht gestört werden möchte.

Der Mythos des In-der-Zone-sein hat dabei nichts mit der ehemaligen DDR zu tun, sondern bezeichnet den Zustand, wenn man sich gerade so richtig gut irgendwo eingearbeitet hat, alle Zusammenhänge frisch verdrahtet im Kopf sind und man tranceähnlich vor sich hin programmiert. Alles fließt wie ein fröhliches Flüsschen dahin, die Finger tippen Codezeile für Codezeile. Wehe, wenn man jetzt stört. Man wird mit bösen Blicken bedacht und muss möglicherweise sogar die eine oder andere Schimpftirade über sich ergehen lassen, wie man es wagen könne, die Programmiererin jetzt, gerade jetzt, AUSGERECHNET JETZT in ihrem Flow zu stören. Sie wird schwer seufzen, das Anliegen zur Kenntnis nehmen und dann schnell wieder die Kopfhörer aufsetzen, um zurück in ihre Zone zu kommen.

Das haben sich die Programmiererinnen natürlich klug ausgedacht, denn wer weniger gestört wird, wird seltener mit doofen Aufgaben belästigt werden und darf öfter und länger einfach nur programmieren.

Allerdings sind Unterbrechungen nicht immer zu vermeiden: Erdbeben, Handwerkertermine, Feuersbrünste und Druck auf der Blase muss man als höhere Gewalt akzeptieren. Außerdem ist ununterbrochenes, konzentriertes Arbeiten körperlich ungesund, und irgendwann geht es gar nicht mehr, weil man schon nach kurzer Zeit Schmerzen bekommt, die sich weigern, in ebenso kurzer Zeit wieder zu verschwinden. Wer über 30 ist, wird wissen, was wir meinen oder es übermorgen herausfinden.

Die gute Nachricht ist: Man kann lernen, mit Unterbrechungen zu leben. Die noch bessere Nachricht ist: Wenn man die richtigen Strategien entwickelt, verbessert sich dadurch sogar die Qualität des Codes. Wenn man schon beim Schreiben darauf achtet, dass man nicht erst 263 Verknüpfungen im Hirn zusammenbasteln muss, um den Code zu verstehen, wird das auch die nächste Programmiererin freuen. Und Bugs oder Verbesserungen, die einem während des Codes auf- oder einfallen, behält man eben nicht einfach im Kopf, wo sie bei der kleinsten Erschütterung ins Nichts fallen, sondern notiert sie auf einem Zettel für später.

Solche Vorkehrungen sorgen nicht nur für mehr Frieden am Arbeitsplatz. Manchmal muss man den Code wieder hervorkramen, den man letztes Jahr geschrieben hat – die Auftraggeberinnen wünschen sich Änderungen, oder beim Jahreswechsel ist überraschend alles in Stücke gefallen, weil man nur Jahreszahlen von

0 bis 2014 vorgesehen hat. Auch das pure Verstreichen von Zeit ist eine Unterbrechung, und zwar eine, die sich kaum verhindern lässt.

Aber das Verstreichen von Zeit hat auch Vorteile, und zwar insbesondere, wenn man in dieser Zeit eben nicht programmiert. Es tut dem Gehirn nämlich gar nicht immer gut, stundenlang an einem Problem herumzuknapsen. Zwar mag es sich so anfühlen, als ob man noch nie so tief im Code gesteckt habe wie jetzt gerade, das Gehirn wird dabei aber auch sehr müde und vergisst, kreativ zu denken. Nach einem harten Tag voll vergeblicher Fehlersuche geht man nach Hause und legt sich ins Bett. Am nächsten Morgen weiß man auf einmal, wo das Problem sitzt, und erledigt alles in 15 Minuten. Das Gehirn braucht Pausen vom Denken, sonst mag es nicht mehr abseits des eingeschlagenen Weges nach anderen Möglichkeiten gucken.

Deshalb ist es gut, Programmiererinnen hin und wieder zu stören: Man tut ein gutes, erzieherisches Werk damit. Eventuell trägt man auf diesem Weg sogar dazu bei, die Codequalität auf einem hohen Niveau zu halten. Am besten unterbricht man Programmiererinnen mit dem Hinweis auf Schokoladenkuchen. So ist die Dauer der Verärgerung nur kurz und man sorgt zusätzlich noch für einen ausgeglichenen Serotoninhaushalt aller Beteiligten.

9

Juli 2015: Rule of Three

Sehr alte Leserinnen dieser Kolumne erinnern sich vielleicht noch an die Schreibgeräte der Vergangenheit. Sie hießen »Schreibmaschinen« und sahen ein bisschen so aus wie Computer, nur dass man auf ein weißes Blatt Papier starren konnte statt in eine leere Datei. Diese Geräte hatten einen entscheidenden Nachteil: Sie waren so vorsintflutlich konstruiert, dass lediglich eine einzige App installiert war – Textverarbeitung eben. Ein weiterer Nachteil war die unterentwickelte Möglichkeit, Geschriebenes wieder zu löschen. Copy&Paste gab's ebenfalls nicht. Es waren harte, entbehrungsreiche Zeiten.

Mit Copy&Paste hat der moderne Mensch eine Fähigkeit erworben, die viel Zeit und Mühen erspart. Möchte man ein Textfragment so oder sehr ähnlich noch mal unterbringen, kann man es einfach markieren, kopieren und an der gewünschten Stelle wieder einsetzen. Toll!

Auch Programmiererinnen mögen Copy&Paste, denn oft genug steht man vor einem Problem, von dem man weiß, dass man es irgendwo schon mal gelöst hat. Dann nimmt man sich einfach das entsprechende Stück Code, kopiert es an den gewünschten Ort und ist im besten Fall schon fertig. Im zweitbesten Fall muss man ein paar Variablen ändern oder eine Zeile umschreiben, aber jedenfalls nicht alles noch mal komplett neu machen. Stößt man später noch mal auf das gleiche Problem, dann macht man es einfach

wieder so und hat ein zweites Mal viel Zeit gespart, die sonst fürs Nachdenken und Tippen draufgegangen wäre.

Die Geschichte hat allerdings einen Haken. Üblicherweise stellt sich nämlich heraus, dass man während des Programmierens kleine Fehler gemacht hat. Das passiert, Programmiererinnen wissen das, und das ist auch gar nicht schlimm. Ärgerlich wird es, wenn sich herausstellt, dass der Fehler ausgerechnet in dem Teil des Codes steckt, den man über die Jahre zehn Mal kopiert hat. Wo man vorher zehn Mal Zeit gespart hat, muss man diese jetzt zehn Mal hintereinander aufbringen, um aufzuräumen.

Noch ärgerlicher wird es, wenn man vergessen hat, wo der Code überall steckt. Jetzt kann man nur hoffen, dass die Entwicklungsumgebung eine vernünftige Suchfunktion hat. Und dass man überhaupt eine Entwicklungsumgebung hat, nicht nur einen schäbigen Texteditor. Ansonsten liegt die Wahrscheinlichkeit, dass man an einer der zehn Stellen vergisst, den Fehler zu beheben, bei ziemlich genau hundert Prozent.

Damit es zu solchen Situationen gar nicht erst kommt, empfehlen kluge Menschen wie der Softwareentwickler und Autor Martin Fowler die *Rule of Three*. Diese Regel besagt, dass man Code einmal aus Faulheit kopieren darf, aber nicht öfter. Man darf ihn also zweimal verwenden, dann ist Schluss. Bei der dritten Begegnung mit demselben Problem lagert man die Lösung besser in eine neue Funktion aus. An der Stelle, wo vorher der Code selbst war, steht dann nur noch ein Aufruf für diese Funktion. Wenn sich nun herausstellt, dass sich in genau diesen Codezeilen ein

Fehler versteckte, muss man ihn nur noch exakt einmal korrigieren und kann sicher sein, dass es danach überall funktioniert.

Wie nützlich die *Rule of Three* ist, merkt man daran, dass sie leicht modifiziert auch im Alltag funktioniert: Man läuft besser nicht beim ersten Anflug des Wunsches nach einer Schlagbohrmaschine zum Baumarkt, sondern erst dann, wenn man sich zum dritten Mal eine ausleihen müsste. Auf Briefe von Ämtern muss man nicht gleich beim ersten oder zweiten Mal reagieren – vielleicht kommt das dritte Schreiben ja nie. Auch der Gesundheit kann die Regel dienen, nämlich dann, wenn man sich vornimmt, erst beim dritten Wunsch nach einer Tüte Erdnussflips vom Sofa aufzuspringen. Ist man allerdings ein Mensch, der für drei Erdnussflipswünsche nur vier Sekunden braucht, halten sich die gesundheitlichen Vorteile in Grenzen. In der Softwareentwicklung bleibt die *Rule of Three* trotzdem eine gute Idee.

10

August 2015: Metasyntaktische Variablen

Mit den Platzhaltern ist es so eine Sache. Manchmal erkennt man gleich, dass sie nur Badehandtücher sind, die den Liegestuhl freihalten. Dem X in der Mathematik sieht man mit geringen Vorkenntnissen an, dass es keine Zahl ist, sondern ein anderes Ding, an dessen Aufenthaltsort man eigentlich eine Zahl erwarten würde. Außerdem bekommen viele Menschen bereits in der Schule beigebracht, dass X durch eine Zahl ersetzt werden muss, weil man sonst Punktabzug bekommt. Wenn in einem auszufüllenden Formularfeld im Netz »Telefonnummer« steht, erkennen auch weniger Aufmerksame, dass so ein Wort wahrscheinlich durch eine Zahl ersetzt werden soll, sonst kann niemand anrufen.

Schwieriger wird es, wenn Wörter den Platz für andere Wörter freihalten. Das Problem besteht jetzt darin, zu erkennen, ob das Wort nur ein Platzhalter ist oder bereits das Gemeinte. Allgemein bekannt ist zum Beispiel, dass man bei der Aufforderung »Sag Bescheid, wenn die Milch überkocht« nicht zwingend »Bescheid!« sagen muss (außer natürlich, man ist ein Scherzkeks). Auch seriös veranlagte Menschen tragen manchmal »ihre-domain.de« irgendwo ein, weil es so in der Anleitung steht, und müssen dann den Support anrufen: »Ich hab alles richtig gemacht, aber wenn ich jetzt ihre-domain.de öffne, ist da gar nicht meine Website!« Selbst superseriösen Personen wie den

Autorinnen dieser Kolumne passiert es, dass sie als Titel für eine Wired-Kolumne »Irgendwas mit Code« notieren und später nicht mehr wissen, ob das bereits der Titel sein sollte oder nur eine Aufforderung, noch mal gründlich nachzudenken.

Vor ähnlichen Problemen stehen auch Programmiererinnen gelegentlich, allerdings seltener beim Programmieren selber als beim Lesen oder Schreiben von Codebeispielen, Anleitungen oder Dokumentationstexten. Auch hier gilt es zu unterscheiden zwischen Wörtern, die exakt so im Code stehen müssen und solchen, bei denen man volle Freiheit hat und genauso gut »Warzenschwein« hinschreiben könnte, ohne dass der Code beim Ausführen wilde Fehler produziert.

Aus diesem Grund haben sich Programmiererinnen irgendwann in der Urzeit auf Wörter geeinigt, die man verwenden kann, wenn man Platzhalter braucht. Genauer gesagt: Platzhalter für Variablennamen, die ja ihrerseits Platzhalter sind. Es handelt sich quasi um Liegestuhlhandtuchvertretungshandtücher, unter Fachleuten auch »metasyntaktische Variablen« genannt. So schont man auf der einen Seite das eigene Gehirn, indem man es nicht mit der Suche nach möglichst eindeutigen Platzhalterbezeichnungen belästigt, auf der anderen Seite erkennen zumindest erfahrenere Leserinnen der Codebeispiele, welche Teile des Texts solche Platzhalter darstellen.

Besonders bekannt sind die Wörter »foo«, »bar« und »baz«. (Fortbildungshungrige Leserinnen seien zur Geschichte der Begriffe auf den Wikipediaeintrag »FUBAR« verwiesen, es winken Anekdoten aus dem Krieg.) Die Verfasserin einer Anleitung schreibt

»if foo == bar«, wenn sie gar nichts über Foo oder Bar mitzuteilen hat, sondern den Vergleichsoperator == erklären möchte. Die Leserin der Anleitung übersetzt diese Platzhalternamen beim Schreiben des eigenen Programms dann in Variablennamen: »if number_of_weasels == number_of_weasel_traps«. Beim Ausführen werden die Platzhalter endlich durch echten Inhalt ersetzt. Man erhält die von einem Computer relativ einfach zu beantwortende Frage, ob 100 gleich viel ist wie 99 und freut sich, dass ein Wiesel verschont bleibt.

Auch Nichtprogrammiererinnen müssen ab und zu anderen Menschen signalisieren, dass der hingeschriebene Text nicht der gemeinte ist und eine Werbeanzeige oder Website später noch mit echtem Inhalt gefüllt werden soll. Genau wie in der Softwareentwicklung kommt es dabei immer wieder zu Missverständnissen. Der von der Grafikerin extra unmissverständlich hingeschluderte Fülltext »dieses produkt ist voll super, alle müssen es kaufen, und zwar jetzt gleich!« landet ohne Rückfrage in Broschüren und auf der Firmenwebsite. Ähnlich wie bei *foo* und *bar* lässt sich das Problem auch durch Verwendung des offiziellen Blindtext-Texts »Lorem ipsum dolor sit amet« nicht immer vermeiden: Eine Suche im Netz fördert viele Websites ans Licht, deren Verantwortliche beim Anblick des Blindtexts nur »wohl Italienisch« gedacht und ihn durchgewinkt haben. Die Fähigkeit zur unmissverständlichen Markierung von Metaebenen ist offenbar generell in der menschlichen Kommunikation noch nicht ausgereift. Vielleicht müsste man Blindtext als Schulfach einführen.

11

September 2015: Stromsparender Code

Dass man Energiesparlampen kaufen und ab und zu das Fahrrad nehmen soll, ist allgemein bekannt. Dass auch schlechter Code zu unnötigem Stromverbrauch führt, wird immer wieder unterschlagen. Dabei braucht eine vollständig ausgelastete CPU so viel Strom wie eine 100-Watt-Glühbirne, nämlich ungefähr 100 Watt.

Sitzt man direkt vor dem Rechner, in dem diese CPU steckt, dann macht sich fahrlässige Programmierung schnell bemerkbar: Alles wird langsamer, der Lüfter springt an, man wartet sehr lange, bis man ein Ergebnis zu sehen bekommt. Wenigstens hilft der Rechner auf die Art nebenbei beim Heizen der Wohnung. Es lohnt sich also, schlechten Code lokal und in den Wintermonaten laufen zu lassen. Passiert dasselbe auf einem Rechner, den man in einem Serverzentrum gemietet hat, profitiert man hingegen gar nicht von der Abwärme und wird von den Missständen wenig mitbekommen. Man muss dann selbst, zum Beispiel unter Zuhilfenahme des Unix-Befehls *top*, nachsehen, welche Prozesse das Gerät gerade wie heftig beanspruchen.

Wenn man keinen ganzen Server gemietet hat, sondern nur einen kleinen digitalen Bretterverschlag auf einem großen Gemeinschaftsserver, geht das unter Umständen nicht so einfach. Dann muss man abwarten, bis man vorwurfsvolle Mails von Mitarbei-

terinnen des Webhosting-Unternehmens erhält, weil im Landkreis des Serverzentrums bei jedem Aufruf des suboptimalen Codes die Lichter dunkler werden.

Energiezehrender Code ist selten guter Code und sollte schon allein aus diesem Grund eliminiert werden. Am gefräßigsten sind fahrlässig eingesetzte while-Schleifen, in denen wochenlang Primzahlen durch andere Zahlen dividiert werden oder Regular Expressions das Gesamtwerk von Ryoki Inoue durchsuchen. Nachdem man solche Probleme beseitigt hat, kann man noch ein bisschen nachmessen, an welchen Stellen sich die Laufzeit des Programms verringern ließe. Oder überlegen, ob man wirklich dauernd irgendwelche Verbindungen von A nach B aufbauen muss, obwohl man vielleicht alles, was man von B braucht, einmalig einmal am Anfang von A holen könnte. Schon geht alles schneller und schöner und nebenbei hat man vielleicht mit dem neuen umweltschonenden Code schon mal eine Primel gerettet.

Leider ist der effizientere Code nicht immer auch der verständlichere. Wenn man bei der nächsten Betrachtung der so umweltfreundlichen wie undurchschaubaren Zeilen große Mengen Strom für Kaffee und trostspendende extraheiße Vollbäder verbraucht, ist unterm Strich nichts gewonnen. Zudem muss man überlegen, ob beim Umschreiben des Codes mehr Energie verbraucht wird als nachher voraussichtlich eingespart werden kann. Es ist – wie immer eigentlich – kompliziert.

Wer Code für sehr kleine Geräte wie Arduino oder Raspberry Pi schreibt, ist jedenfalls schon mal auf der umweltfreundlicheren Seite. Oder man verlegt sich auf

Smartphone-Apps, denn erstens brauchen Handys vergleichsweise wenig Strom und zweitens werden die Nutzerinnen sich schnell beschweren, wenn die App innerhalb von fünf Minuten den Akku leersaugt. Aber Vorsicht: wenn man einfach nur alle rechenintensiven Teile der Anwendung auf dem Server laufen lässt, ist wieder nichts gewonnen.

Wenigstens setzen Serverzentren inzwischen oft auf grünen Strom. Man kann also einfach mit einer geschickten Auswahl des Serverstandorts die Umwelt schonen und dafür zum Ausgleich eine zweifelhafte Schleife mehr ins Programm einbauen. Oder trotzdem ordentlich programmieren und dafür einmal unnötig mit dem Auto zum Supermarkt fahren. Jede Dusche verbraucht ein bis zwei kWh, das heißt, dass man außerdem ziemlich viel schlechten Code ausgleichen kann, indem man ihn ungewaschen schreibt. Und manchmal ist schließlich auch Strom übrig. Die Stromunternehmen wissen dann kaum, wohin damit, zum Beispiel nachts[1]. Man tut der Welt praktisch einen Gefallen, wenn man zu diesen Zeiten miesen Code laufen lässt.

---

1 Während wir diese E-Book-Version zusammensuchen, 2024, ist Strom eher mittags übrig, weil es mehr Photovoltaik in Deutschland gibt. Aber die Behauptung stimmte 2015 auch schon nicht mehr. Wahrscheinlich ist sie nur an diese Stelle geraten, weil es im Haushalt von Kathrin Passigs Mutter Nachtspeicherheizungen aus den 1970er Jahren gibt. Damals wurde man wirklich durch billigere Tarife dazu ermutigt, nachts mehr Strom zu verbrauchen, weil der Betrieb von Kohle- und Atomkraftwerken wirtschaftlicher war, wenn sie rund um die Uhr liefen.

12

Oktober 2015: Absichten und Annahmen

Als Programmiererin muss man sich nicht nur Gedanken über die technische Umsetzung von Problemen machen, sondern manchmal auch fachliche Entscheidungen treffen. In Wahrheit sollte dieser Teil der Arbeit zwar von irgendeiner Art von Produktmanagerin übernommen werden, aber gerade bei kleinen Programmierprojekten macht die gemeine Programmiererin vom Design bis zum Test alles selbst und kommt daher um die ein oder andere Entscheidung nicht herum.

Wenn man eine Anwendung mit vielen Eingabefeldern entwickelt, muss man sich zum Beispiel Gedanken über Restriktionen machen. Welche Felder müssen ausgefüllt werden, welche dürfen leer bleiben? Wie verhindere ich unsinnige Eingaben und stelle sicher, dass niemand den 32. Mai 2035 als Geburtstag oder als Postleitzahl eingibt? Dabei treffen wir unter Umständen Entscheidungen, die auf den ersten Blick supervernünftig erscheinen und sich in dem Moment, wenn richtige Menschen das Programm nutzen als sehr, sehr falsch herausstellen. Der Pfad zur Hölle der schlechten Usability ist gepflastert mit gut gemeinten Designentscheidungen.

Generell ist die Idee, gleich bei der Eingabe von Daten darauf hinzuweisen, dass irgendetwas möglicherweise nicht ganz richtig ist, eine gute Idee. Keine Be-

nutzerin will erst ganz am Ende mitgeteilt bekommen, dass sie ganz am Anfang etwas falsch gemacht hat. Am besten merkt man das also schon bei der Eingabe an. Und wäre es nicht sogar noch besser, dafür zu sorgen, dass eine offensichtlich fehlerhafte Eingabe gar nicht möglich ist?

Doch je mehr Restriktionen und Regeln es gibt, desto größer sind auch die Chancen, dass man sich zu sehr in seinen eigenen Vorstellungen, wie die Welt zu sein hat, verliert und anderen Menschen das Leben unnötig schwer macht. Die Vorstellung, dass ein Nachname mindestens zwei oder sogar drei Buchstaben zu haben hat, ist nur solange korrekt, bis ein netter koreanischer Mensch mit dem Nachnamen O kommt und gezwungen wird, irgendeinen Unsinn anzustellen, um seinen Namen ins System zu schummeln.

Im Laufe der Nachbesserungen an diesem System findet man heraus, dass es nicht nur bei Stars üblich ist, auf den Nachnamen zu verzichten, sondern auch in Indonesien. Stars und Menschen mit indonesischen Namen (manche jedenfalls) scheitern an jedem Eingabeformular, das nach westlicher Tradition mindestens einen Vor- und einen Nachnamen als Pflichtfeld vorsieht. Postleitzahlen sollte man sowieso nur validieren, wenn man sehr sicher ist, bis in alle Ewigkeit nur Nutzerinnen aus einem einzigen Land zu haben. Also am besten einfach gar nicht.

So lernt man nach und nach, dass die eigene Vorstellung von wohlgeformten Namen oder Anschriften auf kultureller Ahnungslosigkeit beruht. Irgendwann kommt man zu dem Schluss, dass es wohl besser ist, gar keine Eingaben zu verbieten und höchstens ab und

zu höflich nachzufragen: »Ihre Postleitzahl 4-8-15-16-23-42 sieht ein wenig seltsam aus. Sind Sie sicher?«

Aber es sind nicht nur Adressfelder, die Probleme bereiten. Beim Programmieren von internationalen oder mehrsprachigen Applikationen lernt man erst so richtig, was man alles nicht als gegeben ansehen sollte: Im Französischen ist es korrekt, vor einem Doppelpunkt ein Leerzeichen zu setzen, in der Schweiz müssen Rechnungen mitunter auf 5 Rappen gerundet werden und im Türkischen gibt es mehrere Varianten des Buchstaben I, was bei ungeschickter Anwendung dazu führen kann, dass die gesamte Datenbank kaputt geht, wenn ein türkischer Server im Spiel ist. Um der Brisanz dieses Themas noch mal Nachdruck zu verleihen, sei angemerkt, dass kein einziges dieser Beispiele erfunden ist.

Eine der Kolumnistinnen bestellte einmal ein Zeitschriftenabo im englischsprachigen Ausland. Irgendwo zwischen Adresseingabefeld und Adressaufkleber saß ein hilfreicher Mensch und korrigierte die »Feuerbachstrasse« in eine nicht existierende »Feuerbackstrasse«. Man kann also das bestprogrammierte System der Welt haben und auf sämtliche Kulturkreisverwirrungen vorbereitet sein. Es hilft alles nichts, wenn am Ende der ultimative Datenkorrektor sitzt: Ein echter Mensch mit guten Absichten und falschen Annahmen.